AFRL-IF-RS-TR-2001-241
Final Technical Report
November 2001

COMPILING EXPLICITLY PARALLEL PROGRAMS

University of Berkeley

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. C278

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

COMPILING EXPLICITLY PARALLEL PROGRAMS

Kathy Yelick, Luigi Semenzato, Geoff Pike,
Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy,
Paul Hilfinger, Susan Graham, David Gay,
Phil Colella, and Alex Aiken

Contractor: University of Berkeley
Contract Number: F30602-95-C-0136
Effective Date of Contract: 26 May 1995
Contract Expiration Date: 30 September 1999
Short Title of Work: Compiling Explicitly Parallel Programs
Period of Work Covered: May 95 - Sep 99

Principal Investigator: Susan Graham
Phone: (510) 642-2059
AFRL Project Engineer: Joseph Carozzoni
Phone: (315) 330-7796

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Joseph A. Carozzoni, AFRL/IFTB, 525 Brooks Road, Rome, NY.

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE NOVEMBER 2001	3. REPORT TYPE AND DATES COVERED Final May 95 - Sep 99

4. TITLE AND SUBTITLE COMPILING EXPLICITLY PARALLEL PROGRAMS	5. FUNDING NUMBERS C - F30602-95-C-0136 PE - 62301E PR - C278 TA - 00 WU - 01
6. AUTHOR(S) Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Berkeley 336 Sproul Hall Berkeley California 94720	8. PERFORMING ORGANIZATION REPORT NUMBER N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency 3701 North Fairfax Drive Arlington Virginia 22203-1714 Air Force Research Laboratory/IFTB 525 Brooks Road Rome New York 13441-4505	10. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2001-241
11. SUPPLEMENTARY NOTES Air Force Research Laboratory Project Engineer: Joseph A. Carozzoni/IFTB/(315) 330-7796	
12a. DISTRIBUTION AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report documents the Titanium language and system for high-performance parallel scientific computing. Titanium uses Java as its base, thereby leveraging the advantages of that language and allowing the focus on parallel computing issues. The main additions to Java are immutable classes, multi-dimensional arrays, an explicitly parallel SPMD model of computation with a global address space, and zone-based memory management. The features and design approach of Titanium are discussed, including an application: a three-dimensional adaptive mesh refinement parallel Poisson solver.

14. SUBJECT TERMS Data Parallel Programming, High-Performance FORTRAN, Memory Hierarchy Simulator	15. NUMBER OF PAGES 56
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

TABLE OF CONTENTS

1.	Modifications Not Yet Described	1
2.	Lexical Structure	2
3.	Program Structure	2
4.	New Standard Types and Constructors	2
5.	New Type Constructors	7
6.	Immutable Classes	12
7.	Pointers and Storage Allocation	12
8.	Templates	18
9.	Operating Overloading	20
10.	Processes	21
11.	Odds and Ends	29
12.	Features Under Construction	30
13.	Additions to the Standard Library	30
14.	Various Known Departures from Java	43
15.	Author's Internal Notes	43
16.	Handling of Errors	46

Titanium Language Reference Manual

This document informally describes our current design for the Titanium language. It is in the form of a set of changes to a C/C++/Java dialect; unless otherwise indicated, the reader may assume the syntax and semantics of Java, version 1.1.

[We will keep all versions of this document under source-code control. Please send corrections and additions to the Keeper, referring to the version number of the document you are working from.]

1 Modifications Not Yet Described

We currently intend to add the following features to vanilla Java:

- Foreign-function interfaces.
- A library of shared data types.

In addition, we expect the following modifications to existing features:

- Remove the `Thread` and `ThreadGroup` classes and methods in other classes that produce them.
- Modifications to arithmetic engine.
- Different handling of exceptions in members of a process team.
- Explicit data-layout support.

2 Lexical Structure

Titanium adds the following new keywords to Java:

```
broadcast   foreach     from        immutable   inline
local       op          overlap     partition
sglobal     single      template    within
```

and the following reserved identifiers:

```
Domain      Point       RectDomain
```

3 Program Structure

Types introduced by Titanium are contained in the package `Ti.lang` ('Ti' being the standard chemical symbol for Titanium). There is an implicit declaration

```
import Ti.lang.*;
```

at the beginning of each Titanium program.

The main procedure of a Titanium program must be declared

```
public single static void main(String single [] single args) {
   ...
}
```

4 New Standard Types and Constructors

4.1 Points

The immutable class Point<N>, for N a manifest positive integer constant, is a tuple of N int's. Point<N>s are used as indices into N-dimensional arrays.

Operations. In the following definitions, p and p_i are of type Point<N>; x, k, and k_i are integers;

- p[i], $1 \leq i \leq N$, is component i of p.
- [$k_1, \ldots, k_n$] is a point whose component i is k_i.
- The '`!=`' and '`==`' operators on Point<N> test for mathematically (un)equal tuples.

- `Point<`N`>.all(`x`)` is the `Point<`N`>` each of whose components is x.
- `Point<`N`>.direction(`d, x`)` for $1 \leq |d| \leq N$, is the `Point<`N`>` whose component $|d|$ is $x \cdot \text{sign}(d)$, and whose other components are 0. x defaults to 1.
- The arithmetic operators `+`, `-`, `*`, `/`, applied to two `Point<`N`>`s produce `Point<`N`>`s by componentwise operations. `*` and `/` are also defined between `Point<`N`>`s and (scalar) integers: for p a `Point<`N`>`, s a scalar, and $\oplus$ an arithmetic operator, $p \oplus s =$ $p \oplus$ `Point<`N`>.all(`s`)` and $s \oplus p =$ `Point<`N`>.all(`s`)` $\oplus\ p$. The `/` operator, in contrast to its meaning on two scalar integer operands, rounds toward $-\infty$, rather than toward 0. Divisions by 0 will raise an exception.
- If R is any of `<`, `>`, `<=`, `>=`, or `==`, then $p_0\ R\ p_1$ for `Point<`N`>`s p_0 and p_1, if p_0`[`i`]` R p_1`[`i`]` for all $1 \leq i \leq N$. The expression p_0`!=`p_1 is equivalent to `!(`p_0`==`p_1`)`.
- The expression p_0`.permute (`p_1`)`, where the p_i are `Point<`N`>`s and p_1 is a permutation of $1, \ldots, N$ is the `Point<`N`>` p for which $p[p_1[i]] = p_0[i]$.
- p`.arity` $= N$, and is a manifest constant.
- The expression p`.toString()` yields a text representation of p.
- [...]

4.2 Domains and RectDomains

The type `Domain<`N`>`, for N a manifest positive integer constant, is an arbitrary set of `Point<`N`>`s. The type `RectDomain<`N`>` is a "rectangular" set of `Point<`N`>`s: that is, a set

$$\{p \mid p_0 \leq p \leq p_1, \text{ and for some } x,\ p = p_0 + S{*}x\}$$

where all quantities here are `Point<`N`>`s. S here (a `Point<`N`>`) is called the *stride* of the `RectDomain<`N`>`, p_0 the *origin,* and p_1 the *upper bound.* `RectDomain<`N`>`s are used as the index sets (bounds) of N-dimensional arrays.

Operations: In the following descriptions, N is positive integer, D is a `Domain<`N`>`, R is a `RectDomain<`N`>`, and RD may be either a `Domain<`N`>` or a `RectDomain<`N`>`.

- There is a standard (implicit) coercion from `RectDomain<`N`>` to `Domain<`N`>`.
- RD`.isRectangular()` is true for all `RectDomains` and for any `Domain` that is rectangular.

- A `Domain<`N`>` may be explicitly converted to a `RectDomain<`N`>` using the usual conversion syntax: `(RectDomain<`N`>)` D. It is a run-time error if D is not rectangular.

- If p_0 and p_1 are `Point<`N`>`s, then `[` $p_0 : p_1$ `]` is the `RectDomain<`N`>` with stride `Point<`N`>.all(1)`, origin p_0, and upper bound p_1. If s is also a `Point<`N`>`, $s >$ `Point<`N`>.all(0)`, then `[` $p_0 : p_1 : s$ `]` is the `RectDomain<`N`>` with origin p_0, upper bound p_1, and stride s. Because of the definitions of origin and upper bound at the beginning of this section, it follows that if not $p_0 \leq p_1$, then both `[` $p_0 : p_1$ `]` and `[` $p_0 : p_1 : s$ `]` are empty.

- If i_j, k_j, and s_j, $1 \leq j \leq N$, are `int`s, with $s_j > 0$, then

  `[` $i_1 : k_1 : s_1, \ldots, i_N : k_N : s_N$`]`

  is the same as

  `[ [` $i_1, \ldots, i_N$ `] : [` $k_1, \ldots, k_N$ `] : [` $s_1, \ldots, s_N$`] ].`

  and

  `[` $i_1 : k_1, \ldots, i_N : k_N$`]`

  is the same as

  `[ [` $i_1, \ldots, i_N$ `] : [` $k_1, \ldots, k_N$ `] ].`

- RD`.arity` $= N$, and is a manifest constant.

- The expression RD`.min()` yields a `Point<`N`>` such that RD`.min()[`k`]` is the minimum over all p in RD of p`[`k`]`. Likewise for RD`.max()`. For empty domains, RD`.min()` yields a point all of whose coordinates are `Integer.MAX_VALUE`, and RD`.max()` yields a point all of whose coordinates are `Integer.MIN_VALUE`.

- RD`.boundingBox()` yields

  `[` RD`.min() :` RD`.max() ].`

  For `RectDomains` R, R`.boundingBox()=`R.

- R`.stride()` yields the minimal `Point<`N`>`, $s >$ `Point<`N`>`.all(0) such that R = `[`$p_0 : p_1 : s$`]` for some p_0 and p_1. (This need not be the same as the stride used to construct R in the first place: for example,

  ```
  ([ 0:1:2, 0:1:2 ]).stride()
  ```

is `[1,1]`, not `[2,2]`.)

- RD.`size()` is the cardinality of RD (the number of Points it contains). The predicate RD.`isNull()` is true iff RD.`size()` $= 0$.

- R.`permute(`p_1`)` is the `RectDomain<`N`>` consisting of all points p.`permute(`p_1`)` for p in RD.

- The operations `+`, `-`, and `*` are defined between any combination of `RectDomain<`N`>`s and `Domain<`N`>`s. They stand for union, difference, and intersection of the sets of elements in the operand domains. The intersection of two `RectDomain<`N`>`s yields a `RectDomain<`N`>`. All other operations and operands yield `Domain<`N`>`s.

- For a `Point<`N`>`, p, the expressions $RD{+}p$, $RD{-}p$, $RD{*}p$, and RD/p compute the domains
$$\{d | d = d' \oplus p, \text{ for some } d' \in RD\}$$
where $\oplus$ = `+`, `-`, `*`, or integer division rounded toward $-\infty$ (unlike ordinary integer division, which rounds toward 0). Divisions require that each $p[i]$ be non-zero, and, if RD is a `RectDomain`, that each RD.`stride(`i`)` either be divisible by $p[i]$ or less than $p[i]$; it is an error otherwise. If RD is a `RectDomain`, so is the result.

- The operations `<`, `==`, `!=`, `>`, `<=`, and `>=` are defined between `RectDomains` and between `Domains` and represent set comparisons (`<` is strict subset, etc.).

- The expression RD`.contains(`p`)`, for `Point<`N`>` p, is true iff $p \in RD$.

- The expression R`.accrete(`k`, ` dir`, ` s`)` gives the result of adding elements to R on the side indicated by direction dir, expanding it by $k \geq 0$ strides of size $s > 0$ (all arguments but R are integers). That is, it computes
$$R + (R + \texttt{Point<}N\texttt{>.direction}(dir, s)) + \ldots + (R + \texttt{Point<}N\texttt{>.direction}(dir, s \cdot k))$$
and returns the result (always rectangular) as a `RectDomain`. It is an error if R could not have been constructed with a stride of s in direction dir, so that the resulting set of elements would not be a `RectDomain`. [It is not possible always to get the stride from R, since it is not well-defined when R is degenerate (empty or having only one value for index k of its `Points`.] The argument s may be omitted; it defaults to 1. Requires that $1 \leq |dir| \leq N$.

- The expression R`.accrete(`k`, ` S`)` gives the result of expanding R on all sides by $k \geq 0$, as for the value V computed by the sequence

V = R.accrete(k, 1, S[1]); V = V.accrete(k, -1, S[1]);
V = V.accrete(k, 2, S[2])....

The argument S is a `Point` with the same arity as R. It may be omitted, in which case it defaults to `all(1)`.

- The expression R.`shrink(`k, *dir*`)` gives the result of shrinking R on the side indicated by direction *dir* by $k \geq 0$ strides of size R.`stride(`$|dir|$`)` That is, it computes

  R*(R+`Point<`N`>.direction(`dir, s`)`)*...*(R+`Point<`N`>.direction(`$dir, -k \cdot s$`)`).

  where s is R.`stride(`$|dir|$`)`. Requires that $1 \leq |dir| \leq N$.

- The expression R.`shrink(`k`)` gives the result of shrinking R on all sides by $k \geq 0$ strides, as for the value V computed by the sequence

  V = R.shrink(k, 1).shrink(k, -1).shrink(k, 2)....

- The expression R.`border(`k, *dir*, *shift*`)` consists, informally, of the k-thick ($k \geq 1$) layer of index positions on the side of R indicated by *dir*, shifted by *shift* positions in direction *dir*. Thus,

  R.border(1, *dir*, 0)

  is the layer of cells on the face of R in direction *dir*, while

  R.border(1, *dir*, 1)

  is the layer of cells just over that face (outside the interior of R). More formally, it is

  R.accrete(k, *dir*) - R + Point<N>.direction(*dir*, *shift*-k)

  Requires that $k \geq 0$, $0 < |dir| \leq N$. As shorthand,

  R.border(k, *dir*) = R.border(k, *dir*, 1)

  and

  R.border(*dir*) = R.border(1, *dir*, 1).

- The expression R.`slice(`k`)`, where $0 < k \leq N$, and $N > 1$ is a `RectDomain<`$N-1$`>` consisting of the set

$$\{[p_1, \ldots, p_{k-1}, p_{k+1}, \ldots, p_N] \mid p = [p_1, \ldots, p_n] \in R\}.$$

- The expression D`.RectDomainList()` returns a one-dimensional array (see §5.1) of type `RectDomain<`N`>[1d]` containing zero or more disjoint, non-null domains whose union (treated as `Domain<`N`>`s) is D.

- The expression `Domain<`N`>.toDomain(`X`)`, where X is of type `RectDomain<`N`>[1d]` and has disjoint members, yields a `Domain<`N`>` consisting of the union of the elements of X.

- The expression D.PointList() returns an array of type `Point<`N`>[1d]` of distinct points consisting of all the members of D.

- The expression `Domain<`N`>.toDomain(`X`)`, where X is of type `Point<`N`>[1d]` and has distinct members, yields a `Domain<`N`>` whose members are the elements of X.

- The expression RD`.toString()` yields a text representation of RD.

- The expression `Domain<`N`>.setRegion(`*reg*`)`, where *reg* is a `Region`, dynamically causes *reg* to become the `Region` used for allocating all internal pointer structures used in representing domains (until the next call of `setRegion`). A null value for `reg` causes subsequent allocations to come from garbage-collected storage. Returns the previous value passed to `setRegion` (initially `null`).

Control Structures: The construct

```
foreach (p in RD) S
```

for RD any kind of domain, executes S repeatedly, binding p to the points in R in some unspecified order. The scope of the control variable p is S. It is constant (`final`) within its scope. The control constructs `break` and `continue` function in `foreach` loops analogously to other loops.

5 New Type Constructors

Titanium modifies Java syntax to allow for *grid types* (§5.1), and the qualifiers `local` (§7) and `single` (§10.3.1).

Type:
    QualifiedBaseType ArraySpecifiers$_{opt}$

QualifiedBaseType:
    BaseType Qualifiers$_{opt}$

ArraySpecifiers:
    ArraySpecifiers ArraySpecifier
    ArraySpecifier

ArraySpecifier:
    `[]` *Qualifiers*$_{opt}$
    `[` *IntegerConstantExpression* `d ]` *Qualifiers*$_{opt}$

Qualifiers:
    Qualifier Qualifiers
    Qualifier

Qualifier: `single` | `local`

BaseType:
    PrimitiveType
    ClassOrInterfaceType

where *PrimitiveType* and *ClassOrInterfaceType* are from the standard Java syntax. The grouping of qualifiers with the types they modify is as suggested by the syntax: In *QualifiedBaseType,* the qualifiers modify the base type; the qualifiers in the array specifiers apply to the type resulting from the immediately preceding array specification. Array specifiers apply to their *QualifiedBaseType* from right to left: that is, '`T [1d][2d]`' is "1-D array of (2-D arrays of T)."

The qualifier **single** is outwardly contagious. That is, the type "array of single T" (`T single [...]`) is equivalent to "single array of single T" (`T single [...] single`).

5.1 Arrays

A Java array (object) of length N—hereafter called a *standard array*—is an injective mapping from the interval of non-negative integers $[0, N)$ to a set of variables, called the *elements* of the array. Titanium extends this notion to *grid arrays* or *grids;* an N-dimensional grid is an injective mapping from a `RectDomain<`N`>` to a set of variables. As in Java, the only way to refer to any kind of array object—standard or grid—is through a pointer to that object. Each element of a standard array resides in (is mapped to from) precisely one array[1]. The elements of a grid, by contrast, may be shared among any number

[1]This does *not* mean that there is only one name for each element. If `X` and `Y` are arrays (of any kind), then after '`X=Y;`', `X[p]` and `Y[p]` denote the same variable. That is an immediate consequence of the fact that arrays are always referred to through pointers.

of distinct grids. That is, even if the array pointers `A` and `B` are distinct, `A[p]` and `B[p]` may still be the same object. We say then that `A` and `B` *share* that element. There is no explicit way to release array elements; it is the responsibility of the system to release them when they are no longer accessible.

For a non-grid type T, the type "N-dimensional grid with element type T" is denoted

T`[`N`d]`

where N is a positive manifest integer constant. When that constant is an identifier, it must be separated from the 'd' by a space[2]. To declare a variable that may reference two-dimensional grids of `doubles`, and initialize it to a grid indexed by the `RectDomain<2>` D, one might write

```
double[2d] A = new double[D];
```

For the type "N_0-dimensional grid whose elements are N_1-dimensional grids... whose elements are of type T," we write

T`[`N_0`d] [`N_1`d]`$\cdots$.

There is a reason for this irregularity in the syntax (where one might have expected a more compositional form, in which the dimensionalities are listed in the opposite order). The idea is to make the order of indices consistent between type designations, allocation expressions, and array indexing. Thus, given

```
double[1d][2d] A = new double[D1][D2];
```

we access an element of `A` with

```
A[p1][p2]
```

where `D1` is a `RectDomain<1>`, `D2` a `RectDomain<2>`, `p1` a `Point<1>`, and `p2` a `Point<2>`.

Two array types are assignment compatible only if they are identical. This is a restriction relative to standard Java, which allows assignment conversion of one array-of-reference type to another if the element types are assignment convertible.

Array types are (at least for now) not quite complete Java Objects, in that they may not be coerced to type Object.

[2]It is true that when N is an integer literal, Nd is syntactically indistinguishable from a floating-point constant in Java. However, in this context, such a constant would be illegal.

Operations: In the following, take $p, p_0, \ldots$ to be of type `Point<`N`>`, A to be a grid of some type T `[`N`d]`, and D to be a `RectDomain<`N`>`.

- A.`domain()` is the domain (index set) of A. It is a `RectDomain<`N`>`.
- A`[`p`]` denotes the element of A indexed by p, assuming that p is in A.`domain()`. It is an lvalue—an assignable quantity—unless A is a global array of local references, in which case it is unassignable.
- The expression A`.copy(`B`)` copies the contents of the elements of B with indices in `A.domain()*B.domain()` into the elements of A that have the same index. It is illegal if A and B do not have the same grid type. It is also illegal if A is a global array whose element type has local qualification (it is easy to construct instances in which such a copy would be unsound). It is legal for A and B to overlap, in which case the semantics are as A were first copied to a (new) temporary before being copied to B.
- `A.arity` is the value of A.`domain().arity` when `A` is non-null, and is a manifest constant.
- If $i_1, \ldots, i_N$ are integers, then A`[`$i_1, \ldots, i_N$`]` is equivalent to A`[ [`$i_1, \ldots, i_N$`] ]`. (Syntactic note: this makes `[...]` similar to function parameters; applications of the comma operator must be parenthesized, unlike C/C++)
- The following operations provide remappings of arrays.
  - A`.translate(`p`)` produces a grid, B, whose domain is `A.domain()+`p, such that B`[`p`+`x`]` aliases A`[`x`]`.
  - A`.restrict(`R`)` produces the restriction of A to index set R. R must be a `RectDomain<`N`>`.
  - A`.inject(`p`)` produces a grid, B, whose domain is `A.domain()*`p, such that B`[`p`*`x`]` aliases A`[`x`]`.
  - A`.inject(`p`).project(`p`)` produces A. For other arguments, `project` is undefined.
  - A`.slice(`k, j`)` produces the grid, B, with domain A`.domain().slice(`k`)` such that

    $$B[[p_1, \ldots, p_{k-1}, p_{k+1}, \ldots, p_n]] = A[[p_1, \ldots, p_{k-1}, j, p_{k+1}, \ldots, p_n]].$$

    It is an error if any of the index points used to index A are not in its domain, or if A`.arity()`≤ 1.

  - `A.permute(`p`)`, where p is a permutation of $1, \ldots, N$, produces a grid, B, where A`[`$i_1, \ldots, i_N$`]` aliases B`[`$i_{p[1]}, \ldots, i_{p[N]}$`]`.
  - `A.set(`v`)`, where v is an expression of type T, sets all elements of A to v.
  - [. . .].

- [...].

Overlapping Arrays. As for any reference type in Java, two grid variables can contain pointers to the same grid. In addition, several of the operations above produce grids that reference the same elements. The possibility of such *overlap* does not sit well with certain code-generation strategies for loops over grids. Using only intraprocedural information, a compiler can sometimes, in principle, determine that two grid variables do not overlap, but the problem becomes complicated in the presence of arbitrary data structures containing grid pointers, and when one or more grid variable is a formal parameter.

For these reasons, Titanium has a few additional rules concerning grid parameters:

- Formal grid parameters to a function (including the implicit `this` in the case of methods defined on grids) may not overlap unless otherwise specified—that is, given two formals F_1 and F_2, none of the variables $F_1[p_1]$ may be the same as $F_2[p_2]$. It is an error otherwise.

- The qualifier '`overlap(`F_1`,`F_2`)`' immediately following a method header, where F_1 and F_2 name formal parameters of that method, means that the restriction does not apply to F_1 and F_2. (There is no restriction that F_1 and F_2 have the same type or even that they be grids. When they are not grids of the same type, however, the qualifier has no effect.)

- To specify that two grid variables X and Y do not overlap at some point in a program, the programmer inserts the call

  X`.noOverlap(`Y`)`.

  This is a method on grids that behaves somewhat as if defined

  ```
  public void noOverlap (T[] B) {},
  ```

  which, by the rules above, does nothing and requires that `B` and `this` not overlap. The qualifier "somewhat" is needed in this description because in fact the call is legal regardless of the element type or local qualification of any of the operands.

6 Immutable Classes

Immutable classes extend the notion of Java primitive type to classes. An immutable class is a final class that is not a subclass of Object and whose non-static fields are final. It is declared with the usual class syntax and an extra modifier

```
immutable class C { ... }
```

Non-static fields are implicitly `final`, but should not be explicitly declared as `final`.

Because immutable classes are not subclasses of any other class, their constructors may not call `super()` explicitly, and, contrary to the usual rule for Java classes other than Object, do not do so implicitly either.

There are no type coercions to or from any immutable class, aside from those defined on standard types in the Titanium library (see §4.2). The standard classes `Point<`N`>`, `RectDomain<`N`>`, and `Domain<`N`>` are immutable.

The value `null` may not be assigned to a variable of an immutable class. The initial value of a variable of immutable class `T` is `new T()`[3]. Initialization of the fields in objects of these types follows the rules of Java 1.2.

By default, the operators `==` and `!=` are defined by comparison of the fields of the object. There may be a `void finalize()` method that is called when immutable objects are no longer accessible.

As a consequence of these rules, it is impossible, except in certain pathological programs, to distinguish two objects of an immutable class that contain equal fields. The compiler is free to ignore the possibility of pathology and unbox immutable objects.

The '+' operator (concatenation) on type `String` is extended so that if S is a string-valued expression and X has an immutable type, then S+X is equivalent to S+X`.toString()` and X+S is equivalent to X`.toString()`+S. These expressions are therefore illegal if there is no `toString` method defined for X, or if that method does not return a `String` or `String local`.

7 Pointers and Storage Allocation

[3]This is not quite sufficient, but suffices for non-pathological programs. In standard Java, the effect of accessing a field before it has been properly initialized is well-defined (even when it is illegal!) in such a way as to be easy to implement (i.e., zero out all storage immediately upon allocation). The analogous implementation for a field of immutable class, assuming that objects of that class are unboxed, has a peculiar semantic description: the field is initialized to point to an *uninitialized* object in which all fields have their default values. This object is then initialized at the normal time (i.e., at the time its constructor would have executed had it been of an ordinary class).

7.1 Demesnes: local vs. global pointers

In Titanium, the set of all memory is the union of a set of local memories, called *demesnes* here to give them a veneer of abstraction. Each object resides in one demesne. Each *process* (§10) is associated with one demesne—called simply the *demesne of the process.* A local variable resides in the demesne of the process that allocates that variable. An object created by **new** and the fields within it reside in the demesne of the process that evaluates the **new** expression.

The static types ascribed to variables (locals, fields, and parameters) containing reference values and to the return values of functions that return reference values are either *global* or *local.* A variable having a local type will contain only null or pointers whose demesnes are the same as that of the variable. A pointer contained in a variable with global type may have any demesne. A local variable may be assigned only a value whose type is (statically) local.

Standard Java type designators for reference types denote global types. The modifier keyword `local` indicates a local reference. For examples:

```
Integer i1;                  /* i is a global reference */
Integer local i2;            /* i is a local reference */
int local i3;                /* illegal: int not a reference type */
```

For grids, there is an additional degree of freedom:

```
/* A1 is a global pointer to an array of variables that may reside
 * remotely (i.e., in a different demesne from the variable A1). */
int[1d] A1;
/* A2 is a local pointer to an array of variables that reside
 * locally (in the same demesne as A2). */
int[1d] local A2;

/* A3 is a local pointer to an array of variables that reside
 * locally and contain pointers to objects that may reside
 * remotely. */
Integer [1d] local A3;
/* A4 is a local pointer to an array of variables that reside
 * locally and contain pointers to objects that reside locally */
Integer local [1d] local A4;
/* A5 is a global pointer to an array of variables that may
 * reside remotely and contain pointers to objects that
 * may reside remotely */
Integer [1d] A5;
```

```
/* A6 is a global pointer to an array of variables that may
 * reside remotely and contain pointers to objects that reside
 * in the same demesne as that array of variables. */
Integer local [1d] A6;
```

We refer to a reference type apart from its local/global attribute as its *object type*.

Coercions between reference types are legal if, first, their object types obey the usual Java restrictions on conversion (plus Titanium's more stringent rules on arrays). Second, a reference value may only be coerced to have a local type (by means of a cast) if it is null or denotes an object residing in the demesne of the process performing the coercion. It is an error to execute such a cast otherwise. Coercions from local to corresponding global types are implicit (they extend assignment and method invocation conversion). Global to local coercions must be expressed by explicit casts.

If a field selection `r.a` or `r[a]` yields a reference value and `r` has static global type, then so does the result of the field selection.

All reference classes support the following operations:

`r.creator()` Returns the identity of the lowest-numbered process whose demesne contains the object pointed to by `r`. NullPointerException if `r` is null. This number may differ from the identity of the process that actually allocated the object when multiple processes share a demesne.

`r.isLocal()` Returns true iff `r` may be coerced to a local reference. This returns the same value as type as `r instanceof T local`, assuming `r` to have object type T. On some (but not all) platforms `r.isLocal()` is equivalent to `r.creator() == thisProc()`.

`r.clone()` Is as in Java, but with local references inside the object referenced by `r` set to null. The result is a global pointer. This operation may be overridden.

`r.localClone()` Is the default `clone()` function of Java (a shallow copy), except that `r` must be local.

To indicate that the special variable `this` in a method body is to be a local pointer, label the method local by adding the `local` keyword to the method qualifiers:

```
public local int length () {...}
```

Otherwise `this` is global. If necessary, the value for which a method is invoked is coerced implicitly to be global. A static method may not be local.

Rationale. The purpose of the distinction between local and global references is to improve performance. Dereferencing a global reference requires a test to see whether the referenced datum is accessible with a native pointer; communication is needed if it is not. Local references are for those data known to be accessible with a native pointer. Global references can be used anywhere but local references don't travel well.

7.2 Region-Based Memory Allocation Concepts

Java uses garbage collection to reclaim unreachable storage. Titanium retains this mechanism but also includes a more explicit (but still safe) form of memory management: *region-based* memory allocation.

In a region-based memory allocation scheme, each allocated object is placed in a program-specified *region*. Memory is reclaimed by destroying a region, freeing all the objects allocated therein. A simple example is shown in Figure 7.2. Each iteration of the loop allocates a small array. The call `r.delete()` frees all arrays.

```
class A {
  void f()
  {
    PrivateRegion r = new PrivateRegion();

    for (int i = 0; i < 10; i++) {
      int[] x = new (r) int[i + 1];
      work(i, x);
    }
    try {
        r.delete();
    }
    catch (RegionInUse oops) {
        System.out.println("oops - failed to delete region");
    }
  }

  void work(int i, int[] x) { }
}
```

Figure 1: An example of region-based allocation in Titanium.

A region r can be deleted only if there are no *external* references to objects in r (a reference external to r is any pointer not stored within r). A call to `r.delete()` throws an exception when this condition is violated.

7.3 Shared and Private Regions

There are two kinds of regions: *shared* regions and *private* regions. Objects created in a shared region are called *shared objects*; all other objects are called *private objects*. Garbage-collectible objects are taken to reside in an anonymous shared region. It is an error to store a reference to a private object in a shared object. It is also an error to broadcast or exchange a private object. As a consequence, it is impossible to obtain a private pointer created by another process.

All processes must cooperate to create and delete a shared region, each getting a copy of the region that represents the same, shared, pool of space. The copy of the shared region object created by a process p is called the *representative* of that region in process p (see the `Object.regionOf` method below). Creating and deleting shared regions thus behaves like a barrier synchronization and is an operation with global effects (see §10.3.1).

A region is said to be *externally referenced* if there is a reference to an object allocated in it that resides in

- A live local variables;
- A static field;
- A field of an object in another region.

The process of attempting to delete a region r proceeds as follows:

1. If r is externally referenced, throw a `ti.lang.RegionInUse` exception.
2. Run the `finalize` methods of all objects in r for which it has not been run.
3. If r is now externally referenced, throw a `ti.lang.RegionInUse` exception.
4. Free all the objects in r and delete r.

Garbage-collected objects behave as in Java. In particular, deleting such objects differs from the description above in that finalization does not wait for an explicit region deletion.

7.4 Detailed Specification of Region-Based Allocation Constructs

Shared regions are represented as objects of the `ti.lang.SharedRegion` type, private regions as objects of the `ti.lang.PrivateRegion` type. The signature of the types is as follows:

```
package ti.lang;

final public class PrivateRegion extends Region
{
  public PrivateRegion() { }
  public void delete() throws RegionInUse;
};

final public class SharedRegion extends Region
{
  public sglobal SharedRegion() { }
  public sglobal void delete() throws RegionInUse single;
};

abstract public class Region
{
};
```

The `delete` method attempts to delete a region as described above.

The Java syntax for `new` is redefined as follows (T is a type distinct from `ti.lang.PrivateRegion` and `ti.lang.SharedRegion`):

- `new ti.lang.PrivateRegion()` or `new ti.lang.SharedRegion()`: creates a region containing only the object representing the region itself.
- `new T...`: allocate a garbage-collected object, as in Java.
- `new (expression) T...` creates an object in the region specified by `expression`. The static type of `expression` must be assignable to `ti.lang.Region`. At runtime the value v of `expression` is evaluated. If v is:
  - `null`: allocate a garbage-collected object, as in Java.
  - an object of type `ti.lang.PrivateRegion` or `ti.lang.SharedRegion`: allocate an object in region v.
  - In all other cases a runtime error occurs.

The class `java.lang.Object` is extended with the following method:

```
public final ti.lang.Region local regionOf();
```

This returns the region of the object, or `null` for garbage-collected objects. For shared objects, the local representative of the shared region is returned.

8 Templates

Titanium uses a "Templates Light" semantics, in which template instantiation is a somewhat augmented macro expansion, with name capture and access rules modified as described below.

8.1 Instantiation Denotations

Define

TemplateInstantiation:
    `template` *Name* `"<"` *TemplateActual* { `","` *TemplateActual* }* `">"`
TemplateActual:
    Type | *AdditiveExpression*

where the AdditiveExpression is a ConstantExpression. Resolution of Name is as for type names. [Note: We use AdditiveExpression to prevent non-LALRness with < and > operators.] It is devoutly to be hoped that the bogus `template` keyword can be eliminated from instantiations.

8.2 Template Definition

TemplateDeclaration:
    TemplateHeader ClassDeclaration
    TemplateHeader InterfaceDeclaration
TemplateHeader:
    `template` `"<"` *TemplateFormal* { `","` *TemplateFormal* }* `">"`
TemplateFormal:
    `class` *Identifier*
    BasicType Identifier

The first form of TemplateFormal allows any type as argument; the second allows ConstantExpressions of the indicated type[4].

[4]At the moment, *some* reference types are allowed for the latter case, but I have suppressed this possibility until it is fully implemented and tested.

8.3 Names in Templates

A template belongs to a particular package, as for classes and interfaces. Access rules for templates themselves are as for similarly modified classes and interfaces.

Template instantiations belong to the same package as the template from which they are instantiated. As a result, it is essentially useless to instantiate a template from a different package except with public classes and interfaces[5].

Names other than template parameters in a template are captured at the point of the template definition. Names in a TemplateActual (and at the point it is substituted for in a template instantiation) are resolved at the point of instantiation.

8.4 Template Instantiation

References to TemplateInstantiation are allowed as Types. They are not allowed in `extends` or `implements` clauses. Instantiations that cause an infinite expansion or a loop are compile-time errors. Inside a template, one may refer to the "current instantiation" by its full name with template parameters; or by the template's simple name. The constructor is called by the simple name. Template formals may not be used in `extends` or `implements` clauses.

```
template class List<class T> {
    ...
    List (T head, List tail) {...}

    List tail() { ... }
}
```

or

```
template class List<class T> {
    ...
    List (T head, template List<T> tail) {...}

    template List<T> tail() { ... }
}
```

[5]The rule we *had* agreed on stated that template instantiations had package-level access to all classes and interfaces mentioned in the template actuals, so that the expansion of a template might be illegal according to the usual rules (e.g., the template expansion could reside in package `P` and yet contain fields whose types were non-public members of package `Q`). The rationale for this rule was that otherwise, a package of utility templates would be useless unless one was willing to make public all classes used as template parameters. However, a Certain Implementor found this rule "quite repugnant" and refused to implement it.

8.5 Name Equivalence

For purposes of type comparison, all simple type names and template names that appear as TemplateActuals are replaced by their fully-qualified versions. With this substitution, two type names are equivalent if their QualifiedName parts are identical and their TemplateActuals are identical (identical types or equal values of the same type).

8.6 Type Aliases

[NOTE: This section is as yet unimplemented.] The import clause is extended to include

ImportDeclaration:
    `import` *Identifier* `"="` *Type*`";"`

which introduces Identifier as a synonym for Type in the current compilation. The standard type import

```
import Qualifier.Name;
```

is thus shorthand for

```
import Name = Qualifier.Name;
```

9 Operator Overloading

Titanium allows overloading of the following Java operators:

Unary prefix	`- ! ~`
Binary	`< > <= >= == !=`
	`+ - * / & | ^ % << >> >>>`
Matchfix	`[] []=`
Assignment	`+= -= *= /= &= |= ^= %= <<= >>= >>>=`

If $\oplus$ is one of these operators, then declaring op$\oplus$ produces a new overloading of that operator. No space is allowed between the keyword `op` and the operator name. These methods can be called like normal methods, e.g. `a.op+(3)`. There are no restrictions on the number of parameters, parameter types or result type of operator methods. In addition,

- An expression $E_1 \oplus E_2$, where $\oplus$ is binary, is equivalent to $E_1 . \oplus (E_2)$ as long as E_1 is not of a primitive type.

- An expression $\oplus E_1$, where $\oplus$ is unary, is equivalent to E_1.op $\oplus$ (), as long as E_1 is not of a primitive type.
- An expression $E_0[E_1, \ldots, E_n] = E$ is equivalent to E_0.op[]=($E_1, \ldots, E_n, E$).
- In other contexts, an expression $E_0[E_1, \ldots, E_n]$ is equivalent to E_0.op[]($E_1, \ldots, E_n$).

It is not possible to redefine plain assignment (=). It is possible to redefine the operator assignment methods, but the assignment remains: $E_1 \oplus$=E_2 is equivalent to E_1 = E_1.op$\oplus$=(E_2), except that expression E_1 is evaluated only once.

There is a conflict between Java's description of how the '+' operator works when the right argument is of type String and the description above, which arises whenever the programmer defines op+ with an argument of type String. We resolve this by analogy with ordinary Java overloading of static functions. If class A defines op+ on String, then anA + aString resolves to the user-defined '+', as if resolving a match between an overloaded two-argument function whose first argument is of type Object and one whose first argument is of type A.

10 Processes

A *process* is essentially a thread of control. Processes may either correspond to virtual or physical processors—this is implementation dependent. Each process has a demesne (an area of memory heap; see §7) that may be accessed by other processes through pointers. Each process has a distinct non-negative integer *index*, returned by the function call Ti.thisProc(). The indices of all processes form a contiguous interval of integers beginning at 0.

Process teams. At any given time, each process belongs to a *process team*. The function Ti.myTeam() returns an int[1d] containing the indices of all members of the team in ascending order. Teams are the sets of processes to which broadcasts, exchanges, and barriers apply. Initially, all processes are on the same team, and all execute the main program from the beginning. This team has the index set [0:Ti.numProcs()-1].

In the current version of Titanium, there is only one process team—the initial one. The process-team machinery in accordingly a bit heavier than needed. It will become useful should we decide to introduce the partition construct (see §12.1).

10.1 Interprocess Communication

Exchange. The member function exchange, defined on Titanium array types, allows all processes in a team to exchange data values with each other. The call

A.exchange(*E*)

acts as a barrier (see §10.2) among the processes with indices in `Ti.myTeam()`. Here, E is of some type T and A is a grid of T with an index set that is a superset of Ti.myTeam().`domain()`. When all processes in a team have reached a call to exchange, then, assuming that all their arguments are of the same type, for each i in `Ti.myTeam()`'s domain, element i of each array is set to the argument E supplied by the process indexed by `Ti.myTeam()`[*i*]. It is an error if the processes reach different textual instances of exchange. It is illegal to exchange arrays of local pointers (that is arrays of a type qualified '**local**').

Thus, the code

```
double [1d] single [2d] x = new double[Ti.myTeam().domain()][2d];
x.exchange(new double [D]);
```

creates a vector of pointers to arrays, each on a separate processor, and distributes this vector to all processors.

Broadcast. The `broadcast` statement allows one process in a team to broadcast a value to all others. Specifically,

`broadcast` *E* `from` *p*

where E is an arbitrary value and p is the index of a process on the current process team, acts as a barrier. Process p (only) evaluates E, and when all processes in the team reach the same barrier, the `broadcasts` all return this value of E. It is an error if the processes reach different textual instances of the call. It is an error if the processes do not agree on the value p—in fact, p must be a single-valued expression (see §10.3.1). It is an error for the evaluation of E on processor p to throw an exception (which, informally, would keep one process from reaching the barrier).

10.2 Barriers

The call

```
Ti.barrier();
```

causes the process executing it to wait until all processes in its team have executed the textually identical barrier call. It is an error for a process to execute this call when another process on its team is waiting at a textually distinct barrier call. Each textually distinct occurrence of `partition` and each textually distinct call on exchange waits on a distinct anonymous barrier.

10.3 Checking Global Synchronization

In SPMD programs, some portions of the data and control-flow are identical across all processes. In particular, the sequences of global synchronizations (barriers, broadcasts, etc.) in each process must be identical for the program to be correct. With the aid of programmer declarations, the Titanium compiler performs a (conservative) check for this correctness condition statically. To be able to perform this check, the compiler must know that certain parts of the program's data are replicated across all processes in the current process team, and that this replicated storage contains coherent values everywhere. By *coherent* we mean that values of primitive types are identical, and that values of reference types point to replicated objects that are of the same dynamic type. The programmer must use the type qualifier **single** to indicate what storage must be coherent.

The formal definition of "coherent" is in terms of pairs of storage locations: two storage locations a and b, residing in demesnes r_a and r_b respectively, and containing values of static type t are *consistent* if t is not **single**, or if:

- t is a primitive type: the values of a and b are identical;
- t is a java or titanium array type: a and b have the same bounds, the elements of a reside in r_a, the elements of b reside in r_b, and the corresponding elements are consistent;
- t is an object type (immutable or not): a and b have the same dynamic type, the object referred to by a resides in r_a, the object referred to by b resides in r_b, and corresponding non-static fields of a and b are consistent.

The compiler constrains all Titanium programs as follows. Given any program statement S, and the current process team P, and assuming

- all free variables of S and all static variables of the program have consistent values in all processes in P;
- `this` has a consistent value in all processes in P

then after the execution of S, and assuming that all processes in P terminate:

- all free variables of S and all static variables of the program still have consistent values in all processes in P;
- all processes have executed the same sequence of methods qualified with **single**—in particular this implies that all processes have executed the same sequence of global synchronization operations.

The following properties are important in checking that programs meet this constraint:

- an expression e is *single-valued* if it evaluates to a consistent value in all processes P that start executing e;
- a termination T (exception of type t, a break, continue or return) of a statement S is *universal* if either all processes P that start S terminate abruptly with termination T or no process P that starts S terminates abruptly with termination T—in addition, if the termination is an exception the value of the exception must be consistent in all processes in P;
- a statement has *global effects* if it or any of its substatements: assigns to any storage (variable, field or array element) whose type is t **single**, *may call* a method or constructor which has global effects, or is a `broadcast` expression
- a method has *global effects* if any of the statements of its body have global effects; however, assignments to **single** local variables do not count as global effects;
- a native or abstract method has *global effects* if it is qualified with **sglobal** (which is a new *MethodModifier* that can modify a method or constructor declaration);
- a constructor has *global effects* if any of the statements of its body have global effects; however, assignments to **single** local variables, or non-static **single** fields of the object being constructed do not count as global effects (the destination of these field assignments must be specified as an unqualified name or as **this.**name and the assignment must occur in the body of the constructor);
- a method call $e_1.m_1(\ldots)$ *may call* a method m_2 if m_2 is m_1, or if m_2 overrides (possibly indirectly) m_1

A `catch` clause in a `try` statement and the `throws` list of a method or constructor declaration indicate that an exception is universal by qualifying the exception type with **single**.

10.3.1 Single-valued expressions

In the following, the e_i are expressions, v is a non-static field and vs a static field. The following expressions are single-valued. In these descriptions, all instances of operators refer to built-in definitions; user-defined operators are governed by the same rules as function calls.

- constants;

- `this`;
- variables whose type is t **single**;
- $e_1.v$ if e_1 is single-valued and v is declared single;
- $e_1.vs$ if vs is declared single;
- $e_1[e_2]$ if e_1 and e_2 are single-valued, e_1 is an array (standard or grid), and the type of the elements of e_1 is single;
- $e_1[e_2]$ if e_1 and e_2 are single-valued and e_1 is a Point.
- $e_1 \oplus e_2$, for e_1 and e_2 single-valued and $\oplus$ a built-in binary operator (likewise for unary prefix and postfix operators);
- $(T)e_1$ if T is single;
- e_1 `instanceof` T if e_1 is single-valued;
- $e_1 = e_2$ if e_2 is single-valued;
- $e_1 \oplus = e_2$ if e_1 and e_2 are single-valued and $\oplus =$ a built-in assignment operator;
- $e_1 ? e_2 : e_3$ if e_1, e_2 and e_3 are single-valued;
- $e_0.v(e_1, \ldots, e_n)$ if:
  - e_0 is single-valued
  - e_i is single-valued if the i'th argument of v is declared single;
  - the result of v is declared single;
- $e_0.vs(e_1, \ldots, e_n)$ if:
  - e_i is single-valued if the i'th argument of vs is declared single;
  - the result of vs is declared single;
- `new` $T(e_1, \ldots, e_n)$ if e_i is single-valued if the i'th argument of the appropriate constructor for T is declared single;
- `new` $T[e_1] \ldots [e_n]$ if $e_1, \ldots, e_n$ are single-valued (in this case the type is

  T `single` $[t_1]...[t_n]$

  where t_i is the type of e_i)

- $[e_1, \ldots, e_n]$ if $e_1, \ldots, e_n$ are single-valued;
- $[e_1 : e_2 : e_3]$ if e_1, e_2, e_3 are single-valued (and similarly for the other domain literal syntaxes);
- `broadcast` e_1 `from` e_2 if e_1 is a primitive type, or an immutable type with no reference-type fields

10.3.2 Restrictions on statements with global effects

The following restrictions on individual statements and expressions are enforced by the compiler:

- assignments to a local variable, method or constructor argument, field or array element whose type is single must be with a single-valued expression;
- if a method call $e_0.v(e_1, \ldots, e_n)$ may call method m_1 which has global effects then:
  - e_0 must be single-valued
  - e_i must single-valued if the i'th argument of v is declared single;
- in a method call $e_0.vs(e_1, \ldots, e_n)$ if vs has global effects then e_i must be single-valued if the i'th argument of vs is declared single;
- in an object allocation `new` $T(e_1, \ldots, e_n)$, if the constructor c is qualified with **single** then e_i must be single-valued if the i'th argument of c is declared single;
- in an array allocation `new` $T[e_1]$, if T is an immutable type and the zero argument constructor for T is qualified with **single** then e_1 must be single-valued;
- in `broadcast` e_1 `from` e_2, e_2 must be single-valued

10.3.3 Restrictions on control flow

The following restriction is imposed on the program's control flow: the execution of all statements and expressions with global effects must be controlled exclusively by single-valued expressions. The following rules ensure this:

- an `if` (or `?` operator) whose condition is not single-valued cannot have statements (expressions) with global effects as its 'then' or 'else' branch;
- a `switch` statement whose expression is not single-valued cannot contain statements with global effects;

- a `while`, `do/while` or `for` loop whose exit condition is not single-valued, and a `foreach` loop whose iteration domain(s) are not single-valued cannot contain statements or expressions with global effects;
- if the main statement of a `try` has non universal terminations then its `catch` clauses cannot specify any universal exceptions;
- associated with every statement or expression that causes a termination t is a set of statements S from the current method or constructor that will be skipped if termination t occurs. If the termination is not universal, then S must not contain any statements or expressions with global effects;

The rules for determining whether a termination is universal are essentially identical to the restrictions on statements with global effects: any termination raised in a statement that cannot have global effects is not universal. In addition:

- in `throw` e, the exception thrown is not universal if e is not single-valued;
- in a call to method or constructor v declared to throw exceptions of types $t_1, \ldots, t_n$, exception t_i is universal only if type t_i is **single** and the following conditions are met:
  - Normal method call $e_0.v(e_1, \ldots, e_n)$: if e_0 is single-valued and e_i is single-valued when the i'th argument of v is declared single;
  - Static method call $e_0.vs(e_1, \ldots, e_n)$: if e_i is single-valued when the i'th argument of vs is declared single;
  - Object allocation `new` $T(e_1, \ldots, e_n)$: if e_i is single-valued when the i'th argument of the appropriate constructor for T is declared single;
  - Immutable array allocation `new` $T[e_1] \ldots [e_n]$: if $e_1, \ldots, e_n$ are single-valued

10.3.4 Restrictions on methods and constructors

The following additional restrictions are imposed on methods and constructors:

- If a method is declared to return a single result, then the expression in all return statements must be single-valued. The return terminations must all be universal;
- Including `throws` t **single** in a method or constructor signature does not allow the method or constructor body to throw a non-universal exception assignable to t;
- A method f that overrides a method g must preserve the singleness of the method argument and result types.

10.4 Consistency of Shared Data

The consistency model defines the order in which memory operations issued by one processor are observed by other processors to take effect. Although memory in Titanium is partitioned into demesnes, the union of those demesnes defines the same notion of shared memory that exists in Java. Titanium semantics are consistent with Java semantics in the following sense: the operational semantics given in the Java Language Specification (Chapter 17) correctly implements the behavioral specification given below. We use the behavioral specification here for conciseness and to avoid constraints (or the appearance of constraints) on implementations.

As Titanium processes execute, they perform a sequence of actions on memory. In Java terminology, they may *use* the value of a variable or *assign* a new value to a variable. Given a variable V, we write $use(V, A)$ for a use of V that produces value A and $assign(V, A)$ for an assignment to V with value A. A Titanium program specifies a sequence of memory events, as described in the Java specification:

> [An implementation may perform] a *use* or *assign* by [thread] T of [variable] V ...only when dictated by execution by T of the Java program according to the standard Java execution model. For example, an occurrence of V as an operand of the + operator requires that a single *use* operation occur on V; an occurrence of V as the left-hand operand of the assignment operator (=) requires that a single *assign* operation occur.

Thus, a Titanium program defines a total order on memory events for each process/thread. This corresponds to the order that a naive compiler and processor would exhibit, i.e., without reorderings. The union of these process orders forms a partial order called the *program order*. We write $P(a, b)$ if event a happens before event b in the program order.

During an actual execution, some of the events visible in the Titanium source may be reordered, modified, or eliminated by the compiler or hardware. However, the processes see the events in some total order that is related to the program order. For each execution there exists a total order, E, of memory events from P such that:[6]:

1. $P(a, b) \Rightarrow E(a, b)$ if a and b access the same variable.
2. $P(l, a) \Rightarrow E(l, a)$, if l is a lock or barrier statement.
3. $P(a, u) \Rightarrow E(a, u)$, if u is an unlock or barrier statement.
4. $P(l_1, l_2) \Rightarrow E(l_1, l_2)$, if l_1 and l_2 are locks, unlocks, or barriers.

[6] See author's notes 1 and 2.

5. E is a correct serial execution, i.e.,

   (a) If $E(write(V, A)), read(V, B))$ and there is no intervening *write* to V, then $A = B$.

   (b) If there were n processes in P, then there are n consecutive barrier statements in E for each instance of a barrier.

   (c) If some process T contains an *unlock* of a l, then the preceding *lock* of l in E must also come from T.

   (d) $P(a, b) \Rightarrow E(a, b)$ if a and b both operate on volatile variables.

Less formally, rule 1 says that dependences in the program will be observed. Rules 2 and 3 say that reads and writes performed in a critical demesne must appear to execute inside that demesne. Rule 4 says that synchronization operations must not be reordered. (Titanium extends the Java rules for synchronization, which include only lock and unlock statements to explicitly include barrier.) Rule 5 says that the usual semantics of memory and synchronization constructs are observed. Rule 6 says that operations on volatile variables must execute in order.

As in Java, the atomic unit for assignment and use is a 32-bit value. In other words, the *assign* and *use* operations in the program order are on at most 32-bit quantities; assignment to double or long variables in Titanium source code corresponds to two separate operations in this semantics. Thus, a program with unsynchronized reads and writes of double or long values may observe undefined values from the reads[7].

11 Odds and Ends

Inlining. The `inline` qualifier on a method declaration acts as in C++. The semantics of calls to such methods is identical to that for ordinary methods. The qualifier is simply advice to the compiler. In particular, you should probably expect it to be ignored when applied to abstract methods or interface methods.

The qualifier may also be applied to loops:

```
foreach (p in Directions) inline {
    ...
}
```

This is intended to mean that the loop is to be unrolled completely. It is ignored if `Directions` is not manifest.

[7]See authors' note 3.

12 Features Under Consideration

This section discusses features of Titanium whose implementation we have deferred indefinitely until we can evaluate the need for them.

12.1 Partition

The constructs

partition { C_0 `=>` S_0; C_1 `=>` S_1; ...; C_{n-1} `=>` S_{n-1}; }

and

partition V { C_0 `=>` S_0; C_1 `=>` S_1; ...; C_{n-1} `=>` S_{n-1}; }

divide a team into one or more teams without changing the total number of processes.

The construct begins and ends with implicit calls to `Ti.barrier()`. When all processes in a team reach the initial barrier, the system divides the team into n teams (some possibly empty). All those for which C_0 is true execute S_0. Of the remaining, all for which C_1 is true execute S_1, and so forth. All processes (including those satisfying none of the C_i) wait at the barrier at the end of the construct until all have reached the barrier.

Since the construct partitions the team, it also changes the value of `Ti.myTeam()`, (but not `Ti.thisProc()`) for all processes for the duration of the construct. If supplied, the variable name V is bound to an integer value such that `Ti.thisProc()` = `Ti.myTeam()`[V]. Its scope is the text of all the S_i.

13 Additions to the Standard Library

13.1 Arrays

This syntax is not, of course, legal. We use it simply as a convenient notation. For each type T and positive integer n:

```
template final class T [n d] {
  public static final int single arity = n;

  public RectDomain<n> single domain();

  public T [n d] single translate(Point<n> single p);
  public local T [n d] local single translate(Point<n> single p);

  public T [n d] single restrict(RectDomain<n> single d);
```

```
    public local T [n d] local single restrict(RectDomain<n> single d);

    public T [n d] single inject(Point<n> single p);
    public local T [n d] local single inject(Point<n> single p);

    public T [n d] single project(Point<n> single p);
    public local T [n d] local single project(Point<n> single p);

    public T [n d] single permute(Point<n> single p);
    public local T [n d] local single permute(Point<n> single p);

    // only if n > 1
    public T [(n - 1)d] single slice(int single k, int single j);
    public local T[(n - 1)d] local single slice(int single k, int single j);

    public void set(T value);

    public void copy(T [n d] x);

    public sglobal void exchange(T myValue);
  }
```

13.2 Points

```
template public immutable class Point<int n> {
      public static Point<n> single all(int single x);
      public static Point<n> single direction(int single k, int single x);
      public static Point<n> single direction(int single k);
      public Point<n> single op+(Point<n> single p);
      public Point<n> single op+=(Point<n> single p);
      public Point<n> single op-(Point<n> single p);
      public Point<n> single op-=(Point<n> single p);
      public Point<n> single op*(Point<n> single p);
      public Point<n> single op*=(Point<n> single p);
      public Point<n> single op/(Point<n> single p);
      public Point<n> single op/=(Point<n> single p);
      public Point<n> single op*(int single n);
      public Point<n> single op*=(int single n);
      public Point<n> single op/(int single n);
      public Point<n> single op/=(int single n);
```

```
  public boolean single op==(Point<n> single p);
  public boolean single op!=(Point<n> single p);
  public boolean single op<(Point<n> single p);
  public boolean single op<=(Point<n> single p);
  public boolean single op>(Point<n> single p);
  public boolean single op>=(Point<n> single p);
  public int single op[] (int single x);

  public static int single arity () { return n; }
  public Point<n> single permute (Point<n> single p);
}
```

13.3 Domains

```
template public immutable class Domain<int n> {
  public static final int single arity = n;

  // Domain set relationships
  public Domain<n> single op+(Domain<n> single d);
  public Domain<n> single op+=(Domain<n> single d);
  public Domain<n> single op-(Domain<n> single d);
  public Domain<n> single op-=(Domain<n> single d);
  public Domain<n> single op*(Domain<n> single d);
  public Domain<n> single op*=(Domain<n> single d);

  // Domain boolean relationships
  public boolean single op==(Domain<n> single d);
  public boolean single op!=(Domain<n> single d);
  public boolean single op<(Domain<n> single d);
  public boolean single op<=(Domain<n> single d);
  public boolean single op>(Domain<n> single d);
  public boolean single op>=(Domain<n> single d);

  // Point set relationships
  public Domain<n> single op+(Point<n> single p);
  public Domain<n> single op+=(Point<n> single p);
  public Domain<n> single op-(Point<n> single p);
  public Domain<n> single op-=(Point<n> single p);
  public Domain<n> single op*(Point<n> single p);
  public Domain<n> single op*=(Point<n> single p);
  public Domain<n> single op/(Point<n> single p);
  public Domain<n> single op/=(Point<n> single p);

  // Shape information
  public int single arity() { return n; }
  public Point<n> single lwb();
  public Point<n> single upb();
  public Point<n> single min();
  public Point<n> single max();
  public int single size();
  public boolean single contains(Point<n> single p);
  public RectDomain<n> single boundingBox();
  public boolean single isNull();
  public boolean single isRectangular();
}
```

13.4 RectDomains

```
public immutable class RectDomain<int n> {
  public static final int single arity = n;
  public boolean isRectangular();

  public Domain<n> single op+(RectDomain<n> single d);
  public Domain<n> single op+=(RectDomain<n> single d);
  public Domain<n> single op-(RectDomain<n> single d);
  public Domain<n> single op-=(RectDomain<n> single d);
  public RectDomain<n> single op*(RectDomain<n> single d);
  public RectDomain<n> single op*=(RectDomain<n> single d);

  public RectDomain<n> single op+(Point<n> single p);
  public RectDomain<n> single op+=(Point<n> single p);
  public RectDomain<n> single op-(Point<n> single p);
  public RectDomain<n> single op-=(Point<n> single p);
  public RectDomain<n> single op*(Point<n> single p);
  public RectDomain<n> single op*=(Point<n> single p);
  public RectDomain<n> single op/(Point<n> single p);
  public RectDomain<n> single op/=(Point<n> single p);

  public boolean single op==(RectDomain<n> single d);
  public boolean single op!=(RectDomain<n> single d);
  public boolean single op<(RectDomain<n> single d);
  public boolean single op<=(RectDomain<n> single d);
  public boolean single op>(RectDomain<n> single d);
  public boolean single op>=(RectDomain<n> single d);

  public Domain<n> single op+(Domain<n> single d);
  public Domain<n> single op+=(Domain<n> single d);
  public Domain<n> single op-(Domain<n> single d);
  public Domain<n> single op-=(Domain<n> single d);
  public boolean single op==(Domain<n> single d);
  public boolean single op!=(Domain<n> single d);
  public boolean single op<(Domain<n> single d);
  public boolean single op<=(Domain<n> single d);
  public boolean single op>(Domain<n> single d);
  public boolean single op>=(Domain<n> single d);

  public final static int arity () { return n; }
  public Point<n> single lwb();
  public Point<n> single upb();
  public Point<n> single min();
  public Point<n> single max();
  public Point<n> single stride();
  public int single size();
  public boolean single isNull();
  public RectDomain<n> single accrete(int single k, int single dir,
                                     int single s);
  public RectDomain<n> single accrete(int single k, int single dir);
  public RectDomain<n> single accrete(int single k, Point<n> single S);
  public RectDomain<n> single accrete(int single k); // S = all(1)
  public RectDomain<n> single shrink(int single k, int single dir);
  public RectDomain<n> single shrink(int single k);
  public final Point<n> single permute (Point<n> single p);
  public RectDomain<n> single border(int single k, int single dir,
                                    int single shift);
  public RectDomain<n> single border(int single k, int single dir);
  public RectDomain<n> single border(int single dir);
  public boolean single contains(Point<n> single p);
  public RectDomain<n> single boundingBox();

  // only if n > 1
  public RectDomain<n - 1> single slice(int single k);
}
```

13.5 Reduction operators

The reduction operators are all barriers. That is, each process in the team must execute the same textual instances of these calls (as well as other barriers) in the same sequence.

```
package ti.lang;
class Reduce
{
  /* The result of Reduce.F (E) is the result of applying the binary
   * operator F to all processors' values of E in some unspecified
   * grouping.  The result of Reduce.F (E, k) is 0 or false for all
   * processors other than K, and the result of applying F to all the
   * values of E for processor K.
   *
   * Here, F can be 'add' (+), 'mult' (*),  'max' (maximum),
   * 'min' (minimum), 'and' (&), 'or' (|), or 'xor' (^).
   */

  public static sglobal int add(int n, int single to);
  public static sglobal long add(long n, int single to);
  public static sglobal double add(double n, int single to);
  public static sglobal int single add(int n);
  public static sglobal long single add(long n);
  public static sglobal double single add(double n);

  public static sglobal int mult(int n, int single to);
  public static sglobal long mult(long n, int single to);
  public static sglobal double mult(double n, int single to);
  public static sglobal int single mult(int n);
  public static sglobal long single mult(long n);
  public static sglobal double single mult(double n);

  public static sglobal int max(int n, int single to);
  public static sglobal long max(long n, int single to);
  public static sglobal double max(double n, int single to);
  public static sglobal int single max(int n);
  public static sglobal long single max(long n);
  public static sglobal double single max(double n);

  public static sglobal int min(int n, int single to);
  public static sglobal long min(long n, int single to);
  public static sglobal double min(double n, int single to);
  public static sglobal int single min(int n);
  public static sglobal long single min(long n);
  public static sglobal double single min(double n);

  public static sglobal int or(int n, int single to);
  public static sglobal long or(long n, int single to);
  public static sglobal boolean or(boolean n, int single to);
  public static sglobal int single or(int n);
  public static sglobal long single or(long n);
  public static sglobal boolean single or(boolean n);

  public static sglobal int xor(int n, int single to);
  public static sglobal long xor(long n, int single to);
  public static sglobal boolean xor(boolean n, int single to);
  public static sglobal int single xor(int n);
  public static sglobal long single xor(long n);
  public static sglobal boolean single xor(boolean n);

  public static sglobal int and(int n, int single to);
  public static sglobal long and(long n, int single to);
  public static sglobal boolean and(boolean n, int single to);
  public static sglobal int single and(int n);
  public static sglobal long single and(long n);
  public static sglobal boolean single and(boolean n);

  /* These reductions use OPER.eval as the binary operator, and are
   * otherwise like the reductions above. */

  public static sglobal Object gen(ObjectOp oper, Object o, int single to);
  public static sglobal Object gen(ObjectOp oper, Object o);

  public static sglobal int gen(IntOp oper, int n, int single to);
  public static sglobal int single gen(IntOp oper, int n);

  public static sglobal long gen(LongOp op, long n, int single to);
  public static sglobal long single gen(LongOp oper, long n);

  public static sglobal double gen(DoubleOp oper, double n, int single to);
  public static sglobal double single gen(DoubleOp oper, double n);
}

package ti.lang;
class Scan
```

```
{
  /* Scan.F (E) produces the result of applying the operation F to all
   * values of E for this and lower-numbered processors, according to
   * some unspecified grouping. */

  public static sglobal int add(int n);
  public static sglobal long add(long n);
  public static sglobal double add(double n);

  public static sglobal int mult(int n);
  public static sglobal long mult(long n);
  public static sglobal double mult(double n);

  public static sglobal int max(int n);
  public static sglobal long max(long n);
  public static sglobal double max(double n);

  public static sglobal int min(int n);
  public static sglobal long min(long n);
  public static sglobal double min(double n);

  public static sglobal int or(int n);
  public static sglobal long or(long n);
  public static sglobal boolean or(boolean n);

  public static sglobal int xor(int n);
  public static sglobal long xor(long n);
  public static sglobal boolean xor(boolean n);

  public static sglobal int and(int n);
  public static sglobal long and(long n);
  public static sglobal boolean and(boolean n);

  /* As for the preceding scans, but with the operation F being
   * oper.eval. */
  public static sglobal Object gen(ObjectOp oper, Object o);
  public static sglobal int gen(IntOp oper, int n);
  public static sglobal long gen(LongOp oper, long n);
  public static sglobal double gen(DoubleOp oper, double n);
}

interface IntOp
```

```
{
  int eval(int x, int y);
}

interface LongOp
{
  long eval(long x, long y);
}

interface DoubleOp
{
  double eval(double x, double y);
}

interface ObjectOp
{
  Object eval(Object arg0, Object arg1);
}
```

13.6 Timer class

A Timer (type `ti.lang.Timer` provides a microsecond-granularity "stopwatch" that keeps track of the total time elapsed between start() and stop() method calls.

```
package ti.lang;
class Timer {

  /**  The maximum possible value of secs (). Roughly 1.84e13. */
  public final double MAX_SECS;

  /** Creates a new Timer object for which secs () is initially 0. */
  public Timer ();

  /** Cause THIS to begin counting. */
  public void start();

  /** Add the time elapsed from the last call to start() to the value
   *  of secs (). */
  public void stop();

  /** Set secs () to 0. */
  public void reset();

  /** The count of THIS in units of seconds, with a maximum
   *  granularity of one microsecond. */
  public double secs();

  /** The count of THIS in units of milliseconds, with a maximum
   *  granularity of one microsecond. */
  public double millis();

  /** The count of THIS in units of microseconds, with a maximum
   *  granularity of one microsecond. */
  public double micros();
}
```

13.7 Additional properties

The values of the following properties are available, as in ordinary Java, through calls to `java.lang.System.getProperty`.

runtime.distributed Has the value `"true"` if and only if the Titanium program reading it has been compiled for a platform with a distributed memory architecture, and otherwise `"false"`.

runtime.model Indicates the specific platform that the Titanium program reading it has been compiled for. At the time this documentation was written, the possible values of this property were: `"cray-t3e"`, `"sp2"`, `"split-c"`, `"smp"`, `"tera-thread"`, and `"pthread"`.

13.8 java.lang.Math

The static methods in `java.lang.Math`, with the exception of `java.lang.random`, take `single` arguments and produce `single` results.

13.9 Polling

The operation `Ti.poll()` services any outstanding network messages. This is needed in programs that have long, purely-local computations that can starve the NIC (e.g. a self-scheduled computation). Currently, this operation has an effect only for Split-C and SP2-targets, but is harmless and legal elsewhere.

13.10 Bulk I/O

The library support fast I/O operations on both Titanium arrays and Java arrays. These operations are synchronous (that is, they block the caller until the operation completes).

13.10.1 Bulk I/O for Titanium Arrays

Bulk I/O works through two methods on Titanium arrays: `.readFrom()` and `.writeTo()`. The arguments to the methods are various kinds of file—currently: `RandomAccessFile`, `DataInputStream`, `DataOutputStream`, in `java.io` and their subclasses `BulkRandomAccessFile`, `BulkDataInputStream`, and `BulkDataOutputStream` in `ti.io`. The methods will throw `IllegalArgumentException` if they are given file arguments that do not support bulk I/O.

Consider a Titanium array type, whose elements are of *atomic type*—a primitive type (e.g. int, double, etc.) or an immutable class whose members are all of atomic type. The methods following are defined for this type:

```
/** Perform a bulk read of data into the elements of this
 *  array from INFILE.  The number of elements read will be
 *  equal to domain().size().  They are read sequentially in
 *  row-major order.  Throws java.io.IOException in the
 *  case end-of-file or an I/O error occurs before all
 *  data are read. */
void readFrom (java.io.RandomAccessFile infile)
     throws java.io.IOException;
void readFrom (java.io.DataInputStream infile)
     throws java.io.IOException;

/** Perform a bulk write of data from the elements of this array
 *  to OUTFILE.  The number of elements written will be equal
 *  to domain().size().  They are written sequentially in row-major
 *  order. Throws java.io.IOException in the case of disk full or
 *  other I/O errors. */
void writeTo (java.io.RandomAccessFile outfile)
    throws java.io.IOException;
void writeTo(java.io.DataOutputStream outfile)
    throws java.io.IOException;
```

I/O on partial arrays. To read or write a proper subset of the elements in a Ti array, first use the regular array-selection methods such as `.slice()` and `.restrict()` to select the desired elements, I/O calls on the resultant arrays (these operations are implemented very efficiently without performing a copy).

13.10.2 Bulk I/O for Java Arrays in Titanium

The `BulkDataInputStream`, `BulkDataOutputStream`, and `BulkRandomAccessFile` classes in the `ti.io` package implement bulk, synchronous I/O. They subclass the three classes in `java.io` that can be used for I/O on binary data (so you can still use all the familiar methods), but they add a few new methods that allow I/O to be performed on entire arrays in a single call, leading to significantly less overhead (in practice speedups of over 60x have been observed for Titanium over). These classes only handle single-dimensional Java arrays whose elements have atomic types (see §13.10.1).

```
package ti.io;

public interface BulkDataInput extends java.io.DataInput {
  /** Perform bulk input into A[OFFSET] .. A[OFFSET+COUNT-1] from this
   *  stream.  A must be a Java array with atomic element type.
   *  Requires that all k, OFFSET <= k < OFFSET+COUNT be valid indices
   *  of A, and COUNT>=0  (or throws ArrayIndexOutOfBoundsException).
   *  Throws IllegalArgumentException if A is not an array of appropriate
   *  type.  Throws java.io.IOException if end-of-file or input error
   *  occurs before all data are read. */
  void readArray(Object A, int offset, int count)
      throws java.io.IOException;
  /** Equivalent to readArray (A, 0, N), where N is the length of
   *  A. */
  void readArray(Object primjavaarray)
      throws java.io.IOException;
}

public interface BulkDataOutput extends java.io.DataOutput {
  /** Perform bulk output from A[OFFSET] .. A[OFFSET+COUNT-1] to this
   *  stream.  A must be a Java array with atomic element type.
   *  Requires that all k, OFFSET <= k < OFFSET+COUNT be valid indices
   *  of A, and COUNT>=0  (or throws ArrayIndexOutOfBoundsException).
   *  Throws IllegalArgumentException if A is not an array of appropriate
   *  type.  Throws java.io.IOException if disk full or other output
   *  error occurs before all data are read. */
  void writeArray(Object primjavaarray, int arrayoffset, int count)
      throws java.io.IOException;
  /** Equivalent to writeArray (A, 0, N), where N is the length of
   *  array A. */
  void writeArray(Object primjavaarray)
```

```
      throws java.io.IOException;
}

public class BulkDataInputStream
    extends java.io.DataInputStream
    implements BulkDataInput
{
  /** A new stream reading from IN.  See documentation of superclass. */
  public BulkDataInputStream(java.io.InputStream in);

  public void readArray(Object A, int offset, int count)
       throws java.io.IOException;

  public void readArray(Object A)
       throws java.io.IOException;
};

public class BulkDataOutputStream
    extends java.io.DataOutputStream
    implements BulkDataOutput
{
  /** An output stream writing to OUT. See superclass documentation. */
  public BulkDataOutputStream(java.io.OutputStream out);

  public void writeArray(Object A, int offset, int count)
      throws java.io.IOException;

  public void writeArray(Object A)
      throws java.io.IOException;
};

public class BulkRandomAccessFile
    extends java.io.RandomAccessFile
    implements BulkDataInput, BulkDataOutput
{
  /** A file providing access to the external file NAME in mode
   *  MODE, as described in the documentation of the superclass. */
  public BulkRandomAccessFile(String name, String mode)
      throws java.io.IOException;
  /** A file providing access to the external file FILE in mode
```

```
   *  MODE, as described in the documentation of the superclass. */
  public BulkRandomAccessFile(java.io.File file, String mode)
      throws java.io.IOException;

  public void readArray(Object A, int offset, int count)
       throws java.io.IOException;
  public void readArray(Object A)
       throws java.io.IOException;

  public void writeArray(Object A, int offset, int count)
       throws java.io.IOException;
  public void writeArray(Object primjavaarray)
};
```

14 Various Known Departures from Java

Blank finals. Currently the compiler does not prevent one from assigning to a blank final field multiple times. This minor pathology is sufficiently unimportant that is unlikely to be fixed, but it is best for programmers to adhere to Java's rules.

15 Authors' Internal Notes

These are collected discussion notes on various topics. This section is not part of the reference manual.

15.1 On Consistency

Note 1. We debated whether there should be a single total order E for a given execution or one for every process in the execution. The latter seems to admit cache implementations that are not strictly coherent, since processes may see writes happening in different orders. Our interpretation of the Java semantics is the stronger single serial order, so we have decided to use that in Titanium. This is subject to change if we find a significant performance advantage on some platform to the weaker semantics. Even with the single serial order, the semantics are quite weak, so it is unlikely that any program would rely on the difference. The following example is an execution that would be correct in the weaker semantics, but not in the stronger one – we are currently unable to find a motivating problem in which this execution would arise.

```
    // initially X = Y = 0
```

```
     P1           P2              P3

     X = 2             X = Y            Y = X
     Y = 1

  // Separating and labeling the accesses:

     P1           P2              P3

 (A)     Write X       Read Y   (C)     Read X   (E)
 (B)     Write Y       Write X (D)     Write Y (F)

  // The following execution constitutes an incorrect behavior:

     Read Y (C) returns 1, Read X (E) returns 2,
     X = 2, Y = 1 at the end of execution.
```

We observe that:

- Access (C) consumes the value produced by (B) since it returns 1. The only other candidate is (F). Let us assume that (F) indeed wrote the value 1. That would imply:
  - (D) hasn't taken place yet
  - (E) read 1
  - (A) must have written 1, which is false.
- Similarly (E) consumes the value produced by (A).
- According to P2, since the final value of X is 2:

  ```
  B < C < D < A
  ```

- According to P3, since the final value of Y is 1:

  ```
  A < E < F < B
  ```

Note 2. The Titanium semantics as specified are weaker than Split-C's in that the default is weak consistency; sequential consistency (the default in Split-C) can be achieved through the use of volatile variables. However, this semantics is stronger than Split-C's *put* and *get* semantics, since Split-C does not require that dependences be observed. For example, a *put* followed by a *read* to the same variable is undefined in Split-C, unless there is an intervening *synch.* This stronger Titanium semantics is much nicer for the programmer, but may create a performance problem on some distributed memory platforms. In particular, if the network reorders messages between a single sender and receiver, which is likely if there are multiple paths through the networks, then two writes to the same variable can be reordered. On shared memory machines this will not be an issue. We felt that it was worth trying to satisfy dependences at some risk of performance degradation.

Note 3. The Java specification makes this qualification of non-atomicity only on non-volatile double and long values. It give the impression (although it is not stated) that accesses to 64-bit volatile values are atomic. This seems to confuse two orthogonal issues: the size of an atomic value and the relative order in which operations occur.

15.2 Advantages of Regions [David Gay]

- The memory management costs are more explicit than with garbage collection: there is a predictable cost at region creation and deletion and on each field write. The costs of the reference counting for local variables should be negligible (at least according to the study we did for our PLDI paper, but I am planning a somewhat different implementation). With garbage collection, pauses occur in unpredictable places and for unpredictable durations.
- Region-based memory management is safe.
- I believe that this style of region-based memory management is more efficient than parallel garbage collection. Obviously this claim requires validation.
- When reference-counting regions instead of individual objects two common problems with reference counting are ameliorated: minimal space is devoted to storing reference counts, and cyclic structures can be collected so long as they are allocated within a single region.

15.3 Disadvantages of Regions

- Regions are obviously harder to use than garbage-collection.
- As formulated above, regions will not mesh well with threads: you need to stop all threads when you wish to delete a shared region. Currently this is enforced by including a barrier in the shared region deletion operation - with threads this is no

longer sufficient. There are a number of possible solutions, but none of them seem very good:

- Require `r.delete()` to be called from all threads. This would be painful for the programmers.
- The implementation of `r.delete()` can just stop the other threads on the same processor. However, to efficiently handle local variables containing references I need to know all points where a thread may be stopped (and obviously if these points are "all points in the program" then efficiency is lost). So this solution doesn't seem very good either.

16 Handling of Errors

Technically, in those places that the language specifically says that "it is an error" for the program to perform some action, the result of further execution of the program is undefined. However, as a practical matter, compilers should comply with the following constraints, absent a compelling (and documented) implementation consideration. In general, a situation that "is an error" should halt the program (preferably with a helpful traceback or other message that locates the error). It is not required that the program halt immediately, as long as it does so eventually, and before any change to state that persists after execution of the program (specifically, to external files). Therefore, it is entirely possible that several erroneous situations might be simultaneously pending, and such considerations as which of them to report to the user are entirely implementation dependent.

Erroneous exceptions. In addition to the erroneous conditions described in other sections of this manual, it is an error to perform any action that, according to the rules of standard Java, would cause the implementation to throw one of the following exceptions implicitly (that is, in the absence of an explicit 'throw' statement):

```
ArithmeticException  ArrayStoreException,
ClassCastException,
IndexOutOfBoundsException, NegativeArraySizeException,
NullPointerException
ThreadDeath
VirtualMachineError (and subclasses)
```

☆U.S. GOVERNMENT PRINTING OFFICE: 2002/710-038/10198

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of Information Systems Science and Technology to meet Air Force unique requirements for Information Dominance and its transition to aerospace systems to meet Air Force needs.